Malloc_Lab

20220124 김문겸

이번 Lab은 Dynamic allocation을 직접 구현하는 과제이다. 이를 위해 Dynamic allocation에서 필요한 함수인 malloc, free, realloc 함수에 대한 기능을 구현해야 한다. 이를 위해 본 Lab에서는 mm_init, mm_malloc, mm_free, mm_realloc 총 4가지의 함수를 구현하도록 되어있다. 함수를 구현하기 위해, 본 강의의 교재인 CSAPP의 코드를 참조하여 작성하였음을 미리 알린다.

상수 및 매크로 정의

```
#define WSIZE 4
#define DSIZE 8
#define CHUNKSIZE (1<<12)

#define MAX(x, y) ((x) > (y)? (x) : (y))

#define PACK(size, alloc)  ((size) | (alloc))

#define GET(p)       (*(unsigned int *)(p))
#define PUT(p, val)  (*(unsigned int *)(p) = (val))

#define GET_SIZE(p)  (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)

#define HDRP(bp)       ((char *)(bp) - WSIZE)
#define FTRP(bp)       ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

#define NEXT_BLKP(bp)  ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)))
#define PREV_BLKP(bp)  ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)))
```

CSAPP 교재의 코드를 참고하여 함수 구현에 사용될 상수 및 매크로와 기타 support 함수들을 정의해주었다.
WSIZE: 1 word의 크기
DSIZE: Align을 위한 double word의 크기
CHUNKSIZE: heap 사이즈 늘릴 시 늘릴 크기
MAX(x, y): x와 y 중 더 큰 값
PACK(size, alloc): 블록의 크기와 할당 여부 비트 합치기
GET(p): p 참조
PUT(p, val): p에 val 값을 저장함
GET_SIZE(p): p의 크기를 표현하는 비트만 추출
GET_ALLOC(p): p의 할당 여부를 표현하는 비트만 추출
HDRP(bp): bp의 header 반환
FTRP(bp): bp의 footer 반환
NEXT_BLKP(bp): 현재 블록의 다음 블록 위치로 이동
PREV_BLKP(bp): 현재 블록의 이전 블록 위치로 이동

```
static char *heap_listp = 0;
static void *extend_heap(size_t words);
static void place(void *bp, size_t asize);
static void *find_fit(size_t asize);
static void *coalesce(void *bp);
```

heap_listp: Dynamic Allocation을 진행할 Heap을 가리키는 포인

extend_heap: heap 크기를 늘일 필요가 있을 때 늘리는 함수
place: allocate할 블록의 크기와 위치를 받아 할당하는 함수
find_fit: allocate할 블록의 위치를 찾는 함수
coalesce: free 시 previous 및 next 블록의 할당 여부에 따라 coalesce을 수행하는 함수

mm_init

```
int mm_init(void)
{

    if ((heap_listp = mem_sbrk(4*WSIZE)) == (void *)-1)
        return -1;
    PUT(heap_listp, 0);
    PUT(heap_listp + (1*WSIZE), PACK(DSIZE, 1));
    PUT(heap_listp + (2*WSIZE), PACK(DSIZE, 1));
    PUT(heap_listp + (3*WSIZE), PACK(0, 1));
    heap_listp += (2*WSIZE);

    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
        return -1;
    return 0;
}
```

동적할당을 시작하기 전 heap_listp를 세팅하는 함수이다. 가장 처음에는 alignment을 위해 0을 넣어 padding 영역을 배치한다. 이후 차례대로 prologue heaer, footer, epilogue header 영역을 배치한다. 그 다음 heap 영역을 가리키는 heap_listp 포인터를 footer 자리로 옮겨 이후의 작업이 알맞은 공간에서 진행되도록 세팅한다.

mm_malloc

```c
void *mm_malloc(size_t size)
{
    size_t asize;
    size_t extendsize;
    char *bp;

    if (heap_listp == 0){
        mm_init();
    }

    if (size == 0){
        return NULL;
    }


    asize = ALIGN(size + DSIZE);


    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }

    extendsize = MAX(asize,CHUNKSIZE);

    if ((bp = extend_heap(extendsize/WSIZE)) == NULL){
        return NULL;
    }


    place(bp, asize);
    return bp;
}
```

mm_malloc은 malloc, 즉 동적할당을 진행하는 함수이다. 우선 먼저 heap_listp가 0이라면, 처음 그대로라면 mm_init을 통해서 초기화를 먼저 진행해준다. 이후 할당하려는 크기가 0이라면, 아무것도 할당하지 않으므로 NULL을 반환한다. 예외 처리 이후 ALIGN 매크로를 통해서 할당하려는 블록 크기의 alignment를 맞춰준다. 그다음 find_fit을 통해서 할당 가능한 크기의 free block을 탐색한다. 발견한다면 place를 통해 할당을 진행하고, 찾지 못했다면 heap의 크기를 늘려 할당할 수 있게끔 만든 다음 새롭게 할당한다.

mm_free

```c
oid mm_free(void *bp)

    size_t size;

    size = GET_SIZE(HDRP(bp)); // bp 크기
    PUT(HDRP(bp), PACK(size, 0)); // free할 블록의 header 0으로 변환
    PUT(FTRP(bp), PACK(size, 0)); // free할 블록의 footer 0으로 변환
    coalesce(bp); // previous/next 블록이 free인 경우 coalesce
```

mm_free에서는 할당된 블록을 할당 해제하는 과정을 수행하는 함수이다.
next block과 previous block의 경우에 따라 병합이 필요할 수 있기 때문에 coalesce 함수를 통해 해당 기능을 수행한다.

mm_realloc

```c
void *mm_realloc(void *ptr, size_t size)
{
    size_t oldsize;
    void *newptr;

    if(size == 0) { // 할당할 크기가 0이므로 아무것도 할당x, 즉 free로 처리
        mm_free(ptr);
        return 0;
    }

    if(ptr == NULL) { // ptr이 NULL이라면 새롭게 할당
        return mm_malloc(size);
    }

    if((newptr = mm_malloc(size)) == NULL){ // newptr에 새롭게 동적할당, 이때 할당
        return 0;
    }



    oldsize = GET_SIZE(HDRP(ptr)); // realloc 전 기존 블록의 크기

    if(size > oldsize){ // 기존 블록 크기가 realloc한 새로운 블록의 크기보다 작다면
        memcpy(newptr, ptr, oldsize); // 기존 블록 크기까지 복사
    }
    else{
        memcpy(newptr, ptr, size); // 아니면 끝까지 복사
    }
    mm_free(ptr); // 새로운 블록 할당 후 기존 블록 free

    return newptr;
}
```

mm_realloc에서는 기존에 할당된 블록의 크기를 변경하는 기능을 수행한다. mm_realloc에서는 realloc을 진행할 블록의 위치인 ptr과, 변경할 크기인 size를 argument로 받는다.

만약 size가 0이라면 새로운 할당 블록의 크기가 0이라는 의미, 즉 free로 처리한다. 만약 ptr이 NULL이라면, 그냥 새로운 malloc을 수행하듯이 진행한다. 이러한 예외처리가 끝난 후, newptr에 realloc을 진행할 새로운 동적할당을 진행한다.

만약 새로운 블록의 크기보다 기존 블록이 작을 경우, 기존 블록에서 원래 크기만큼 값을 복사해서 새로운 블록에 넣는다. 하지만 새로운 블록이 더 작을 경우, 기존 블록에서 새로운 블록만큼만 복사해서 새로운 블록에 저장한다. 이러한 작업이 끝나면 realloc이 마무리되었으므로, 기존 블록을 할당 해제한다.

extend_heap

```c
static void *extend_heap(size_t words)
{
    char *bp;
    size_t size;


    size = (words % 2) ? (words+1) * WSIZE : words * WSIZE; // align 작업


    if ((bp = mem_sbrk(size)) == -1)  // mem_sbrk로 heap 공간 확장
        return NULL;

    PUT(HDRP(bp), PACK(size, 0)); // 새로 확장된 heap 공간 0으로 초기화
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));

    return coalesce(bp); // 새로운 free block이 생겼으므로 다시 coalesce

}
```

extend_heap은 heap을 늘여야 할 필요가 발생할 경우 수행되는 함수이다. 우선 argument로 words, 즉 확장시킬 heap의 크기를 받고, size에 words를 alignment를 맞춘 값을 저장한다. 이후 size 크기만큼 mem_sbrk를 통해 heap을 확장한다. 이후 확장한 공간을 free block으로 초기화한 다음, coalesce를 진행하여 필요한 병합과정을 진행한다.

coalesce

```c
static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

    if (prev_alloc && next_alloc) { // next, previous 둘 다 allocated일 경우
        return bp; // 바로 리턴
    }

    else if (prev_alloc && !next_alloc) { // next만 free block일 경우
        size += GET_SIZE(HDRP(NEXT_BLKP(bp))); // next블록 크기만큼 free block의 크

        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));
    }

    else if (!prev_alloc && next_alloc) { // previous만 free block일 경우
        size += GET_SIZE(HDRP(PREV_BLKP(bp))); // previous 크기만큼 free block 확장

        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }

    else { // next, previous 둘 다 free block일 경우
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp))); // p
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }
    return bp;
}
```

coalesce는 특정 블록을 free시킬 때 previous 블록이나 next block이 free 상태일 경우, 주변 free block과 병합하여 하나의 free block으로 만드는 과정을 수행하는 함수이다. 우선 prev_alloc을 previous block의 할당 여부, next_alloc을 next block의 할당 여부를 나타내도록 저장했다.
coalesce는 총 4가지 case가 존재한다. 1번째 case는 next와 previous 둘 다 allocated인 경우, 2번째는 previous 블록만 allocated인 경우, 3번째는 next 블록만 allocated인 경우, 마지막으로 4번째는 두 블록 모두 free block인 경우이다. 이러한 경우들을 prev_alloc과 next_alloc을 통해 나누어주고, 각각에 해당되는 케이스에 따라 병합과정을 수행한다.

find_fit

```
static void *find_fit(size_t asize) // 강의 pdf dynamic memory allocation - basic cor
{

    void *bp;

    for (bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) { // bp를 heap
        if (!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) { // 만약 할당되지
            return bp; // 해당 블록의 위치를 반환
        }
    }
    return NULL;
}
```

find_fit은 CSAPP의 코드뿐만 아니라 강의 PDF의 first fit 구현 코드를 참조하였다. heap의 가장 처음인 heap_listp부터, 블록 크기가 0이 아닐 때까지, 즉 heap 공간 내 블록들에 대해서 NEXT_BLKP 매크로를 통해 하나씩 순차적으로 조회하면서 할당하고자 하는 블록 크기에 적합한 free block을 first fit 방식으로 탐색한다. 만약 발견할 경우, 해당 블록의 위치를 반환하고, 발견하지 못할 경우 NULL을 반환한다.

place

```
static void place(void *bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp)); // 할당하고자하는 위치의 블록 크기

    if ((csize - asize) >= (2*DSIZE)) {  //할당하려는 크기와 할당하려는 블록의 크기 차
        PUT(HDRP(bp), PACK(asize, 1));
        PUT(FTRP(bp), PACK(asize, 1));
        bp = NEXT_BLKP(bp);

        PUT(HDRP(bp), PACK(csize-asize, 0));
        PUT(FTRP(bp), PACK(csize-asize, 0));
    }
    else {
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }
}
```

place 함수는 find_fit을 통해 찾은 충분한 크기의 free block 위치에 asize 크기만큼 allocate를 진행하는 함수이다. 이때 두 가지 경우가 존재한다. 첫 번째는 free block에 asize크기만큼 할당한 후 남은 공간이 작을 경우, 두 번째는 남은 공간이 다른 블록을 allocate할 수 있을 만큼 큰 경우이다. 후자의 경우 fragmentation을 줄이기 위해 공간을 splitting해주어야 한다. place 함수에서는 그러한 빈 공간이 DSIZE의 두배보다 크거나 같을 경우 splitting을 진행하도록 구현하였다.

이렇게 교재의 코드를 참조하여 작성한 동적할당 코드를 evaluate해본 결과,

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   99%    5694  0.007338   776
 1       yes   99%    4805  0.007408   649
 2       yes   55%   12000  0.084600   142
 3       yes   55%    8000  0.084127    95
 4       yes   51%   24000  0.272067    88
 5       yes   51%   16000  0.273002    59
 6       yes   99%    5848  0.006686   875
 7       yes   99%    5032  0.006670   754
 8       yes   66%   14400  0.000106135849
 9       yes   66%   14400  0.000105137667
10       yes   99%    6648  0.011344   586
11       yes   99%    5683  0.011342   501
12       yes  100%    5380  0.008311   647
13       yes  100%    4537  0.008428   538
14       yes   92%    4800  0.006586   729
15       yes   92%    4800  0.006451   744
16       yes   92%    4800  0.006033   796
17       yes   92%    4800  0.006049   794
18       yes   27%   14401  0.071231   202
19       yes   27%   14401  0.072478   199
20       yes   34%   14401  0.002242  6422
21       yes   34%   14401  0.002236  6442
22       yes   66%      12  0.000000 40000
23       yes   66%      12  0.000000 60000
24       yes   89%      12  0.000000 40000
25       yes   89%      12  0.000000 60000
Total          75%  209279  0.954838   219

Perf index = 45 (util) + 15 (thru) = 59/100
```

59점이 나왔다. 좋지 않은 점수라고 판단하였고, 수업 시간에 배운 다양한 방법들을 고민하며 성능을 높일 수 있는 방안을 생각해보았다.

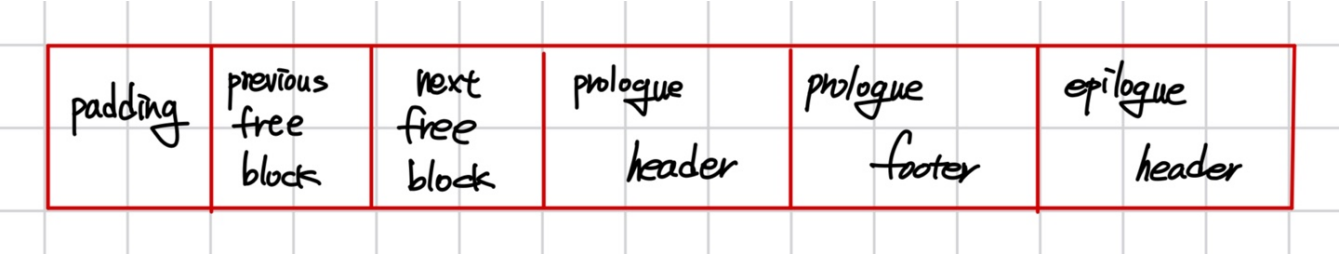
강의 교재를 참고하여 작성한 본 코드는 implicit method를 기반으로 구현하였다. 하지만 implicit 이외에도 explicit method, segregated free list 등 다양한 매소드들이 존재함과 함께 해당 매소드들의 구조를 수업시간에 배웠다. 따라서 implicit보다는 성능이 좋고, 수업시간에 자세하게 배웠던 Explicit method를 적용하여 구현해보기로 했다.

Explicit method

allocated 여부에 상관없이 모든 블록을 참조했던 implicit과 달리 explicit은 free block만을 모은 리스트에서 free block을 조회하기 때문에 훨씬 더 빠른 free block 탐색이 가능하다. Explicit method를 구현하기 위해 previous block과 next block을 가리키는 포인터를 저장할 2개의 블록을 추가적으로 만들어주었다. 본 method에서는 free block 관리를 LIFO로 수행할 것이다.

```
PUT(heap_listp, 0); //align을 위한 padding
PUT(heap_listp + (1*WSIZE), 0);  // previous free 블록을 가리키는 포인터
PUT(heap_listp + (2*WSIZE), 0);  // next free 블록을 가리키는 포인터
PUT(heap_listp + (3*WSIZE), PACK(DSIZE, 1)); //prologue header
PUT(heap_listp + (4*WSIZE), PACK(DSIZE, 1)); //prologue footer
PUT(heap_listp + (5*WSIZE), PACK(0, 1)); //epilogue header
```

따라서 블록 구조는 아래와 같이 된다.



그 다음으로, LIFO 방식을 채택했기 때문에 free block list의 가장 처음을 가리킬 root를 설정해주었다.

```
for (bp = GET(root); bp != NULL; bp = GET(next_list(bp))) { //
```

bp를 root의 주소로 설정한 후, bp가 NULL이 아닐 때까지, 즉 free list의 전체를 다 돌때까지 하나씩 넘어가면서 탐색을 진행한다.

그 다음으로 explicit free list는 블록 전체가 아닌 free list만을 확인하기 때문에, find_fit 함수를 free list만 조회하도록 for 문의 bp 초기화를 heap_listp에서 root로 바꾸어주었다.

free list가 생겼으니 mm_free 또한 free list를 고려하는 코드로 변경되어야 한다고 판단하여, 그 다음으로 free list를 변경했다.

```
void mm_free(void *ptr)
{
    size_t size;
    size = GET_SIZE(HDRP(ptr)); // bp 크기
    PUT(HDRP(ptr), PACK(size, 0)); // free할 블록의 header 0으로 변환
    PUT(FTRP(ptr), PACK(size, 0)); // free할 블록의 footer 0으로 변환

    PUT(next_list(ptr), 0); // 이전 블록 연결 끊기
    PUT(prev_list(ptr), 0); // 다음 블록 연결 끊기
    coalesce(ptr); // previous/next 블록이 free인 경우 coalesce
}
```

free된 블록의 이전 블록과 다음 블록의 연결을 끊어야 하므로 PUT함수를 통해 이전 블록을 가리키는 워드와 다음 블록을 가리키는 워드를 0을 만들어주었다. (이전 블록과 다음 블록을 이어주는 작업은 coalesce 함수에서 구현하였다.)

이때 previous free block을 가리키는 곳과 next free block을 가리키는 곳을 참조하는 과정은 이후에도 많이 사용될 것 같아 prev_list와 next_list 함수로 만들어두었다.

```
static void* prev_list(void* bp) { // 이전 블록으로 이동
    return (void*)bp;
}

static void* next_list(void* bp) { // previous free block 다음 워드가 nex
    return (void*)bp + WSIZE;
}
```

free가 진행된 후 다음으로 coalesce 과정이 진행되므로, 해당 과정을 수행하는 coalesce 함수를 다시 만들어보았다.

coalesce 과정은 Explicit method에서도 4가지 경우가 존재한다. 1번째 case인 next, previous block 모두 allocated일 경우, coalesce를 진행하지 않아도 되므로 바로 root에 넣어 free list에 넣어준다. 해당 과정을 구현한 코드는 아래와 같다.

```
void* rootmp; // root의 주소 저장

if (prev_alloc && next_alloc) { // next, previous 둘 다 allocated일 경우
    rootmp = GET(root); // 원래 root 주소 get
    PUT(prev_list(rootmp), bp); // 원래 root의 previous free block을 새로운 free block으로
    PUT(next_list(bp), rootmp); // 새로운  free 블록의 다음 블록을 원래 root로 저장
    PUT(root, bp); // root를 새로운 free block로 변경
    return bp; // 바로 리턴
}
```

해당 코드는 이후에도 사용될 것 같아 freemake 함수로 정의하여 구현해주었다.

```
static void freemake(void* bp)
{
    void* rootmp= GET(root); //free list의 시작 포인터를 받아온다
    if (rootmp != NULL) //null이 아니면
    {
        PUT(prev_list(rootmp), bp); // 그 이전을 가리키는 포인터에 넣고자 하는 포인터를 저장하고
    }
    PUT(next_list(bp), rootmp); // 원래 시작과 연결해주어서 링크드 리스트에서 (새로운 포인터) -> (기존 시작
    PUT(root, bp); // free list 시작을 업데이트 해준다
}
```

위는 새롭게 free block을 LIFO 구조에 따라 free list에 추가하는 과정을 구현한 freemake 함수이다.

그런데 주변 block이 free block일 경우 기존 free list에서 free인 주변 블록을 삭제한 후 새롭게 free된 블록과 coalesce 하여 새로 리스트에 넣어준다. 해당 과정을 구현한 코드는 아래와 같다.

```
PUT(prev_list(next_list(GET(NEXT_BLKP(bp)))), GET(next_list(bp))); //previous의 next
PUT(next_list(next_list(GET(NEXT_BLKP(bp)))), GET(prev_list(bp))); // next의 previou
PUT(prev_list(bp), 0);
PUT(next_list(bp), 0);
```

그런데 위 코드는 치명적인 문제가 있었다. 만약 '병합할 주변 free block'의 주변이 없을 경우 에러가 발생한다는 것이다. 따라서 병합할 free block의 1) next, previous 둘다 존재하는 경우, 2) next만 존재하는 경우, 3) previous만 존재하는 경우, 4) 둘다 없는 경우 이렇게 4가지로 나누어 삭제 루틴을 진행하였다.

```
static void connection(void* bp){
    void *prev = prev_list(bp);
    void *next = next_list(bp);

    if(GET(prev) && GET(next)){ // 둘 다 블록이 존재할 경우
        PUT(prev_list(GET(next)), GET(prev)); //previous의 next 블록이 기존 블록의 next가 되도록
        PUT(next_list(GET(prev)), GET(next)); // next의 previous 블록이 기존 블록의 previous가 되도
        PUT(next_list(bp), 0);
        PUT(prev_list(bp), 0);

        return;
    }

    else if(!GET(prev) && GET(next)){ // next 블록만 존재할 경우, 즉 root일 경우
        PUT(prev_list(GET(next)), 0);
        PUT(root, GET(next));
        PUT(next_list(bp), 0);
        PUT(prev_list(bp), 0);

        return;
    }

    else if(GET(prev) && !GET(next)){ // previous 블록만 존재할 경우, 즉 맨 뒤일 경우
        PUT(next_list(GET(prev)), GET(next));
        PUT(next_list(bp), 0);
        PUT(prev_list(bp), 0);

        return;
    }
    // 주변에 아무것도 없을 경우
    PUT(root, GET(next));
    PUT(next_list(bp), 0);
    PUT(prev_list(bp), 0);

    return;


}
```

해당 루틴을 connection이라는 함수로 구현하였는데, 4가지 경우에 따라 분류하여 각각의 과정을 코드로 작성하였다. 이 결과 공통되는 부분이 있어 간결하게 바꾸어주었다.

```
static void connection(void* bp){
    void *prev = prev_list(bp);
    void *next = next_list(bp);

    if(GET(prev) && GET(next)){ // 둘 다 블록이 존재할 경우
        PUT(prev_list(GET(next)), GET(prev)); //previous의 next 블록이 기
        PUT(next_list(GET(prev)), GET(next)); // next의 previous 블록이 기
    }

    else if(!GET(prev) && GET(next)){ // next 블록만 존재할 경우, 즉 root
        PUT(prev_list(GET(next)), 0);
        PUT(root, GET(next));
    }

    else if(GET(prev) && !GET(next)){ // previous 블록만 존재할 경우, 즉 
        PUT(next_list(GET(prev)), GET(next));

    }
    else {// 주변에 아무것도 없을 경우
        PUT(root, GET(next));
    }
    PUT(next_list(bp), 0);
    PUT(prev_list(bp), 0);

    return;


}
```

그렇게 구현한 coalesce 코드는 아래와 같다.

```
static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));


    if(prev_alloc && next_alloc){ // next, previous 둘 다 allocated일 경우
        freemake(bp);
    }
    else if (prev_alloc && !next_alloc) { // next만 free block일 경우
        connection(NEXT_BLKP(bp));

        size += GET_SIZE(HDRP(NEXT_BLKP(bp))); // next블록 크기만큼 free block의 크기 증가시킴

        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size, 0));

        freemake(bp);
    }

    else if (!prev_alloc && next_alloc) { // previous만 free block일 경우
        connection(PREV_BLKP(bp));

        size += GET_SIZE(HDRP(PREV_BLKP(bp))); // previous 크기만큼 free block 확장

        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);

        freemake(bp);
    }

    else{ // next, previous 둘 다 free block일 경우
        connection(PREV_BLKP(bp));
        connection(NEXT_BLKP(bp));

        size += GET_SIZE(HDRP(PREV_BLKP(bp))) + GET_SIZE(FTRP(NEXT_BLKP(bp))); // prev, next

        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);

        freemake(bp);
    }
    return bp;
}
```

그 다음으로 place를 구현해보았다.

```
static void place(void *bp, size_t asize)
{
    size_t csize = GET_SIZE(HDRP(bp)); // 할당하고자하는 위치의 블록 크기
    connection(bp); // bp 공간에 할당할 예정이므로 더이상 free block이 아님, 따라서 free block list

    if ((csize - asize) >= (2*DSIZE)) {  //할당하려는 크기와 할당하려는 블록의 크기 차이가 Align 2
        PUT(HDRP(bp), PACK(asize, 1)); // allocate
        PUT(FTRP(bp), PACK(asize, 1)); // allocate
        bp = NEXT_BLKP(bp); // 다음 블록, 즉 할당 후 남는 공간
        PUT(next_list(bp), 0); // free list 초기화
        PUT(prev_list(bp), 0);

        PUT(HDRP(bp), PACK(csize-asize, 0)); // 블록 정보 업데이트
        PUT(FTRP(bp), PACK(csize-asize, 0)); // 블록 정보 업데이트

        coalesce(bp); // 새롭게 free block이 나왔으므로 coalesce 진행
    }
    else { // 크지 않다면, spliiting 진행하지 않고 바로 진행
        PUT(HDRP(bp), PACK(csize, 1));
        PUT(FTRP(bp), PACK(csize, 1));
    }
}
```

우선 먼저 bp에 새롭게 할당이 될 예정이므로, connection 함수를 통해서 bp를 free block list에서 삭제하는 과정을 추가한다.
그 다음 split 과정이 필요하다면, split 후 남는 공간에 대해, 해당 공간은 새롭게 형성된 free block이므로 coalesce 과정을 수행해야 한다. 처음에 coalesce과정을 빼먹고 실행한 결과, 세그멘테이션 오류가 발생했다.

남은 함수는 mm_realloc과 extend_heap이다. 먼저 extend_heap을 수정해보았다.

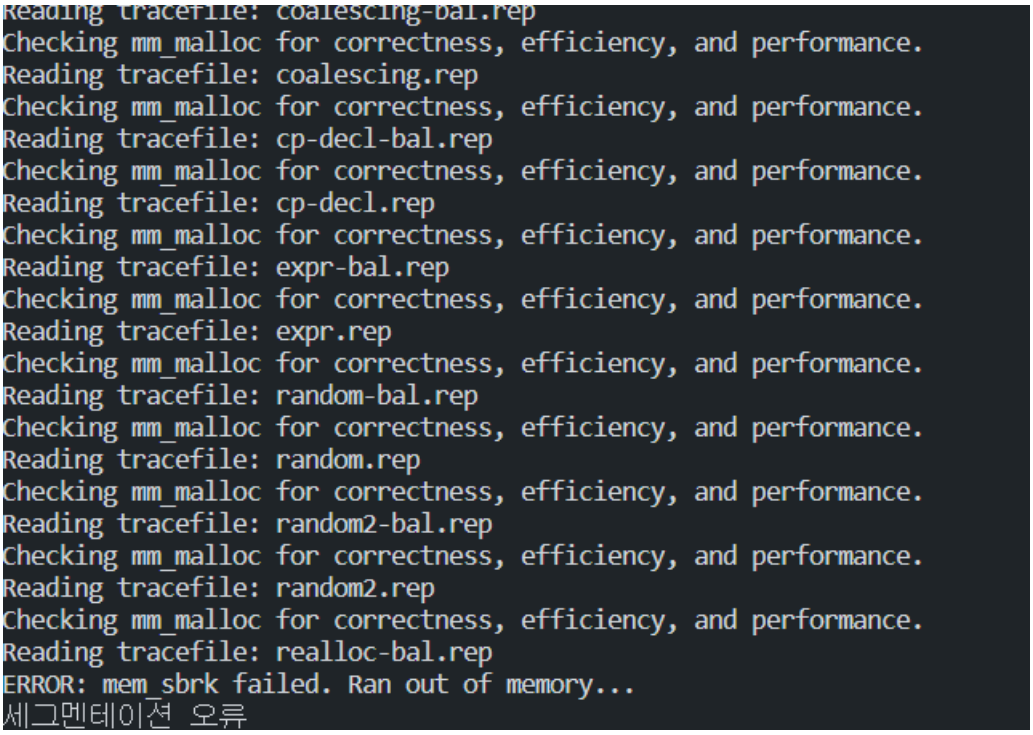
```
static void *extend_heap(size_t words)
{
    char *bp;
    size_t size;


    size = (words % 2) ? (words+1) * WSIZE : words * WSIZE; // align 작업

    if ((bp = mem_sbrk(size)) == -1)  // mem_sbrk로 heap 공간 확장
        return NULL;

    PUT(HDRP(bp), PACK(size, 0)); // 새로 확장된 heap 공간 0으로 초기화
    PUT(FTRP(bp), PACK(size, 0));
    PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1));

    PUT(next_list(bp), 0);
    PUT(prev_list(bp), 0);

    return coalesce(bp); // 새로운 free block이 생겼으므로 다시 coalesce

}
```

여기서는 PUT(next_list(bp), 0)와 PUT(prev_list(bp), 0)을 추가했다. 즉 Explicit method를 적용함으로써 추가로 만든 free list의 next와 prev를 0으로 초기화하는 작업을 추가한다.

mm_realloc의 경우 다음과 같다.

```
void *mm_realloc(void *ptr, size_t size)
{
    size_t oldsize;
    void *newptr;
    void *next;
    size_t expsize;
    size_t sum;

    if(size == 0){ // 할당할 크기가 0이므로 아무것도 할당x, 즉 free로 처리
        mm_free(ptr);
        return 0;
    }

    if(ptr == NULL){ // ptr이 NULL이라면 새롭게 할당
        return mm_malloc(size);
    }

    oldsize = GET_SIZE(HDRP(ptr)); // realloc 전 기존 블록의 크기
    next = NEXT_BLKP(ptr);

    expsize = size + 2*WSIZE; // size + 2*WSIZE, Explicit으로 인해 추가된 next, previous block 포인터 공


    if(expsize > oldsize){ // 기존 블록 크기가 realloc한 새로운 블록의 크기보다 작다면

        newptr = mm_malloc(expsize);
        place(newptr, expsize); // newptr에 expsize 크기만큼 할당
        memcpy(newptr, ptr, expsize); // 기존 정보 복사

        mm_free(ptr); // 기존 블록 할당 해제
        return newptr;

    }
    else{
        return ptr;
    }
}
```

Explicit method의 경우 implicit에 비해 previous free block과 next free block을 가리키는 포인터를 저장할 블록 2개가 증가했으므로 2 word를 추가한다.



그런데 이렇게 수정한 결과 mem_sbrk가 실패했다는 오류가 발생했다. 이는 더 이상 heap의 크기를 늘릴 수 없을 만큼 늘어났다는 것이다. 빈 공간을 재활용하지 않고 계속 heap을 만들어 사용했다는 의미가 된다. 이를 해결하기 위한 방안은 fragmentation을 줄이는 것이라고 생각했고, 이를 줄일 수 있는 방안을 생각해보았다.

1번째로 문제를 발견했던 것은 바로 직전의 realloc이었다. realloc에서 새로운 블록을 할당할 때, 기존 블록 크기가 새로운 블록 크기보다 작다면 바로 mm_malloc을 통해 새로운 공간을 동적할당한다. 이 경우 충분한 공간을 계속 찾지 못하는 최악의 상황이 되면 mem_sbrk를 계속 진행하게 된다. 따라서 위와 같은 에러가 발생한 것으로 추측할 수 있다. 따라서 최대한 fragmentation을 줄일 수 있는 방안을 생각해보던 중, 기존 블록 크기가 새로운 블록 크기보다 작다면 주변의 블록과 합쳐서 새로운 블록을 할당할 수 있는 충분한 공간을 배치하는 방안을 생각해보았다. 따라서 이와 같이 구현하였다.

```c
void *mm_realloc(void *ptr, size_t size)
{
    size_t oldsize;
    void *newptr;
    void *next;
    size_t expsize;
    size_t sum;

    if(size == 0){ // 할당할 크기가 0이므로 아무것도 할당x, 즉 free로 처리
        mm_free(ptr);
        return 0;
    }

    if(ptr == NULL){ // ptr이 NULL이라면 새롭게 할당
        return mm_malloc(size);
    }

    oldsize = GET_SIZE(HDRP(ptr)); // realloc 전 기존 블록의 크기
    next = NEXT_BLKP(ptr);

    expsize = size + 2*WSIZE; // size + 2*WSIZE, Explicit으로 인해 추가된 next, previous block 포인터


    if(expsize > oldsize){ // 기존 블록 크기가 realloc한 새로운 블록의 크기보다 작다면
        if(!GET_ALLOC(HDRP(next)) && (oldsize + GET_SIZE(HDRP(next)) >= expsize)){ // 만약 다음블록이

            connection(next); // 이전 block과 병합 진행:
            PUT(HDRP(ptr), PACK(oldsize + GET_SIZE(HDRP(next)), 1));
            PUT(FTRP(ptr), PACK(oldsize + GET_SIZE(HDRP(next)), 1));

            return ptr;
        }

        else{ // 합치지 못할 경우 새로 할당
            newptr = mm_malloc(expsize);
            place(newptr, expsize); // newptr에 expsize 크기만큼 할당
            memcpy(newptr, ptr, expsize); // 기존 정보 복사


            mm_free(ptr); // 기존 블록 할당 해제
            return newptr;
        }
    }
    else{
        return ptr;
    }
}
```

우선 previous free block과 next free block을 가리키는 포인터를 저장할 블록 2개를 추가로 넣었으므로, 새롭게 할당할 사이즈에 2 word 크기만큼 추가한다. 그 크기로 똑같이 진행한다. 기존 블록 크기가 realloc한 새로운 블록의 크기보다 작다면, next block까지 활용할 수 있는지 check한다. next block이 free block이고, next block과 현재 블록의 크기를 합쳤을 때 새롭게 할당할 블록의 사이즈를 커버할 수 있다면 다음블록과 기존 블록을 합친 후 해당 위치에 할당한다.
만약 next block과 현재블록을 합쳤을 때도 기존 블록이 들어가지 않는다면, 새로운 공간을 찾아서 동적 할당한다.
만약 기존 블록크기가 realloc한 새로운 블록의 크기보다 크다면, 그대로 해당 공간을 사용한다.

그 결과 segmentation fault 없이 무사히 잘 evaluation이 실행되었고,

```
Results for mm malloc:
trace  valid  util     ops      secs  Kops
 0       yes   89%    5694  0.000192 29687
 1       yes   89%    4805  0.000177 27162
 2       yes   55%   12000  0.002237  5365
 3       yes   55%    8000  0.002207  3625
 4       yes   51%   24000  0.002192 10950
 5       yes   51%   16000  0.002088  7663
 6       yes   92%    5848  0.000130 44916
 7       yes   92%    5032  0.000115 43605
 8       yes   66%   14400  0.000147 97893
 9       yes   66%   14400  0.000147 98294
10       yes   94%    6648  0.000270 24640
11       yes   94%    5683  0.000251 22596
12       yes   96%    5380  0.000209 25692
13       yes   96%    4537  0.000200 22685
14       yes   88%    4800  0.000414 11586
15       yes   88%    4800  0.000414 11597
16       yes   85%    4800  0.000454 10582
17       yes   85%    4800  0.000443 10845
18       yes   97%   14401  0.000117123085
19       yes   97%   14401  0.000118122562
20       yes   38%   14401  0.000117122980
21       yes   38%   14401  0.000115125226
22       yes   66%      12  0.000000 40000
23       yes   66%      12  0.000000 60000
24       yes   89%      12  0.000000 60000
25       yes   89%      12  0.000000 60000
Total          77%  209279  0.012754 16409

Perf index = 46 (util) + 40 (thru) = 86/100
```

86점이라는 결과를 얻었다. 만점은 아니지만 이전에 비해 높은 점수를 얻었기에 만족하고 여기서 이번 lab을 마무리하였다.